How we built a Geo-Distributed Database with low latency
About Me

Ming Zhang (张明)
Research and Develop Engineer, PingCAP
Committer of TiDB SQL-Infra SIG
Github: @djshow832
Agenda

- The geographic problem in databases
- What is TiDB?
- How does Geo-Distribution work in TiDB?
- Q&A
The geographic problem in databases
Multiple Geographic Regions

Why multiple geographic regions?
- Improve access locality to achieve lower latency
- Tolerate the failure of an available zone (AZ) or an entire region
Multiple Geographic Regions

Tradeoff
● Latency
● Consistency level

MySQL Replication
● Asynchronous replication: low latency & lower consistency level
● Semisynchronous replication: high latency & higher consistency level
● Group replication: not optimized for multiple geographic regions
What is TiDB
What is TiDB

Open-source distributed NewSQL database for hybrid transactional and analytical processing (HTAP) which speaks MySQL protocol

**Horizontal Scalability**
Transparent scale-out without architectural complexity

**High Availability**
Auto-failover and self-healing to ensure business continuity

**Strongly Consistent**
Full ACID transactions at scale in distributed environments

**MySQL Compatibility**
Without changing MySQL application code in most cases
TiDB OLTP Architecture

Stateless SQL Layer

- TiDB node 1
- TiDB node 2
- TiDB node 3

Distributed Key-Value Storage Engine

- TiKV node 1
  - Store 1
    - Region 1
    - Region 3
    - Region 4
  - Region 1
- TiKV node 2
  - Store 2
    - Region 1
    - Region 2
    - Region 3
- TiKV node 3
  - Store 3
    - Region 2
    - Region 4
    - Region 3
- TiKV node 4
  - Store 4
    - Region 1
    - Region 2
    - Region 4

Placement Driver (PD)

- PD node 1
- PD node 2
- PD node 3

Balance / Failover

- Raft Group
Data organization

What is a region?
- A table is split into regions
- Each region is a bunch of continuous KV pairs
- Region meta: [start key, end key]

<table>
<thead>
<tr>
<th>TiKV Node</th>
<th>Local RocksDB instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store 1</td>
<td>t1_r1 v1</td>
</tr>
<tr>
<td>Region 1</td>
<td>t1_r2 v2</td>
</tr>
<tr>
<td>Region 2</td>
<td>... ...</td>
</tr>
<tr>
<td>Region 3</td>
<td>t5_r1 ...</td>
</tr>
<tr>
<td>Region 4</td>
<td>t5_r10 ...</td>
</tr>
<tr>
<td></td>
<td>t1_i1_1_1 ...</td>
</tr>
<tr>
<td></td>
<td>t1_i1_2_2 ...</td>
</tr>
<tr>
<td></td>
<td>... ...</td>
</tr>
<tr>
<td></td>
<td>t1_i6_1_3 ...</td>
</tr>
<tr>
<td></td>
<td>... ...</td>
</tr>
</tbody>
</table>

Region 1
Region 2
Region 3
Region 4
Write and Read

Raft group
- All replicas of a region form a raft group

Raft roles
- Leader (only one)
- Follower
- Learner (optional)

Write
- Data is written to the leader as logs
- The leader replicates logs to followers and learners
- Logs replicated to the majority of voters are committed

Read
- Read from the leader
Transaction Model

Two-phase commit (2PC)
1. TiDB requests a start timestamp as the identifier of the transaction: start_ts
2. TiDB prewrites data to TiKV
3. TiDB requests a commit timestamp for the transaction before commit: commit_ts
4. TiDB commits data on TiKV with commit_ts

MVCC
- commit_ts is attached to each version of data
- Snapshot isolation
How does Geo-Distribution work in TiDB
Geographic Problems in TiDB

AZ1
- TiKV node 1 (follower)
- TiKV node 2 (leader)
- PD leader
- PD follower
- TiDB node 1

AZ2
- Client
- Write request
- TiKV node 2 (follower)
- TiKV node 3 (leader)
- TiKV node 4 (follower)
Geographic Problems in TiDB

AZ1

- TiDB node 1
- PD follower
- TiKV node 1
  - follower
  - ... (multiple entries)
- TiKV node 2
  - leader
  - ... (multiple entries)

AZ2

- Client
- Read request
- TiDB node 2
- PD follower
- TiKV node 3
  - follower
  - ... (multiple entries)
- TiKV node 4
  - follower
  - ... (multiple entries)

Connections:
1. Client to TiDB node 2
2. TiKV node 1 to TiKV node 2
3. TiKV node 2 to TiKV node 3
4. TiKV node 2 to TiKV node 4
Geographic Problems in TiDB

Bidirectional replication
- Split the system into separate TiDB clusters, which reside in different AZ
- Clusters replicate to each other by synchronization tools
- Suffer from maintaining multiple clusters

What do we want?
- Maintain only one TiDB cluster
- Write and read with low latency
- High availability
Placement Policy

What is placement policy

- Define the placement and replica count of raft roles through SQL

Scenarios

- Place data across regions to improve access locality
- Limit data within its national border to guarantee data sovereignty
- Place latest data to SSD and history data to HDD
- Place the leader of hot data to a high-performance TiKV instance
- Increase the replica count of more important data
Placement Policy

A use case
- A user management system
- Users are distributed over the world
- Users visit their own information through the system

Deployment
- Two data centers located in two AZ
- Applications connect to the nearest data center
- Users typically connect to the nearest application

Solution
- Store each user information in the nearest data center according to their location
- Applications request user information from the local data center
**Placement Policy**

**Configuration**
- Group components by AZ
- Mark instances with the same `zone` label

![Diagram showing TiDB and TiKV nodes in the west zone]

- **PD leader**
  - `zone='west'`
- **PD follower**
  - `zone='west'`
- **TiKV node 1**
  - **leader**
  - `zone='west'`
- **TiKV node 2**
  - **follower**
  - `zone='west'`
Placement Policy

Statements

CREATE TABLE user (id BIGINT AUTO_INCREMENT,
   name VARCHAR(100), country VARCHAR(100),
   PRIMARY KEY(country, id))

PARTITION BY LIST COLUMNS(country) (
   PARTITION east VALUES IN('china', 'japan', 'singapore'),
   PARTITION west VALUES IN('usa', 'canada', 'england', 'france')
);

ALTER TABLE user ALTER PARTITION east
   ALTER PLACEMENT POLICY ROLE=leader CONSTRAINTS='["+zone=east"]';
ALTER TABLE user ALTER PARTITION west
   ALTER PLACEMENT POLICY ROLE=leader CONSTRAINTS='["+zone=west"]';
Placement Policy

**west**
- TiDB node 1
  - PD leader
  - PD follower
- TiKV node 1
  - follower
  - leader
- TiKV node 2
  - follower
  - leader

**east**
- Client
- TiDB node 2
- TiKV node 3
  - leader
  - follower
- TiKV node 4
  - follower

Leaders of partition `west`:
- TiKV node 1
- TiKV node 2

Leaders of partition `east`:
- TiKV node 3
- TiKV node 4
Placement Policy

How does it work
● User defines the placement policies by SQL
● TiDB generates placement rules and send them to PD
● PD schedules data according to the placement rules

Each rule mainly contains:
● Key range: the data range of a table or partition
● Raft role: the raft role to be placed
● Constraints: the labels which TiKV instances match
Local Transaction

Problems of requesting timestamps
- Request timestamps from the PD leader
- The request crosses different AZ
- Request 2 timestamps for each transaction: start_ts & commit_ts
Local Transaction

**Timestamp allocators**
- Elect a local timestamp allocator (PD leader or PD follower) for each AZ
- PD leader is the global timestamp allocator
Local Transaction

What are local transactions?
- Local transactions request timestamps from the local timestamp allocator
- Avoid crossing AZ latency
Global Transaction

Limitations of local transactions
- Clock bias exists among local timestamp allocators, so accessing the same data violates linearizability
- Local transactions can only visit local data
- Data placement is defined through placement policies

Why global transactions?
- When a transaction crosses different AZ
- When a transaction accesses global data, such as metadata

What are global transactions?
- Global timestamp is allocated from the global timestamp allocator
- Conform to linearizability: previous local timestamp < global timestamp < later local timestamp
Global Transaction

How do global transactions work?
1. The global timestamp allocator collects max timestamps allocated by all local timestamp allocators (local_ts)
2. The global timestamp allocator calculates Tmax: Tmax = max (local_ts, ...) + 1

Returns the max timestamp allocated by AZ2 (local_ts2)

Tmax = max(local_ts1, local_ts2) + 1
Global Transaction

How do global transactions work?
3. The global timestamp allocator broadcasts $T_{max}$ to all local timestamp allocators
4. Local timestamp allocators update their local timestamp starting points $local_{ts}$

![Diagram showing the process of global transactions]

- **AZ1**
  - PD leader
  - PD follower

- **AZ2**
  - PD follower
  - Sends $T_{max}$ to local timestamp allocator for AZ2

- **AZ1**
  - PD leader
  - PD follower

- **AZ2**
  - PD follower
  - $local_{ts2} = \max(T_{max}, local_{ts2})$
Global Transaction

How do global transactions work?
5. The global timestamp allocator allocates timestamps from Tmax
6. Local timestamp allocators allocate timestamps from local_ts2
Local & Global Transaction

Local transaction limitations
- Data must be bound to one AZ
- A local transaction can only read / write the data from the current AZ

Global transaction limitations
- Able to access any data
- Cross region 3 times for allocating a global timestamp
- Typically only used for accessing data that not bound to any AZ, such as metadata
Local Stale Read

Why local stale read?
- When data is not bound to AZ, placement policies and local transactions are not applicable
- Followers can also be read
- Sometimes strong consistency is not a must

What is local stale read?
- Read the local replica, including followers
- Read stale data, thus do not guarantee linearizability
- Guarantee snapshot isolation
Local Stale Read

AZ1

TiDB node 1
PD follower

PD leader

TiKV node 1
follower

TiKV node 2
leader

AZ2

Client
Read request

TiDB node 2

TiKV node 3
follower

TiKV node 4
...
Local Stale Read

Example

```sql
SELECT * FROM users JOIN orders WHERE users.id=orders.user_id
AS OF TIMESTAMP '2021-05-01 12:00:00';
```

Semantic

- The transaction reads local replicas
- The transaction reads the same snapshot of `users` and `orders`
- The snapshot is no staler than '2021-05-01 12:00:00'
- Read as new data as the local AZ has

Restriction of start_ts

- All data for the snapshot has been replicated to the current AZ
- No inflight commits which will update the snapshot later
Local Stale Read

Maintain safe_ts

- safe_ts is the commit_ts of the latest data which is replicated to all replicas
- Each raft group maintains a region-wide safe_ts
- TiKV maintains a store-wide safe_ts
- TiKV reports store-wide safe_ts to TiDB periodically
- TiDB maintains an AZ-wide safe_ts

\[ \text{safe}_ts = \min(\text{safe}_ts1, \text{safe}_ts2) \]

\[ \text{safe}_ts1 = \min(\text{safe}_ts \text{ of all regions}) \]
Local Stale Read

Determine start_ts
- TiDB determines start_ts locally
- start_ts is no staler than the user-specified timestamp
- start_ts is no staler than the AZ-wide safe_ts

\[
\text{start_ts} = \max(\text{safe}_ts, \text{stale}_ts)
\]
Summary

When data is bound to AZ
- Use placement policy to define placement of data
- Use local transactions to access local data
- Use global transactions to access global data

When data is not bound to AZ & not need strong consistency
- Use local stale read to read local data
More Resources

- Website: https://pingcap.com/
- GitHub: https://github.com/pingcap/tidb
- Twitter: https://twitter.com/PingCAP
- Slack: #everyone on Slack
About Us

PingCAP is a software service provider committed to delivering one-stop enterprise-grade database solutions.

TiDB is an open-source, distributed New SQL database for elastic scale and real-time analytics.
Thank You!

Q & A